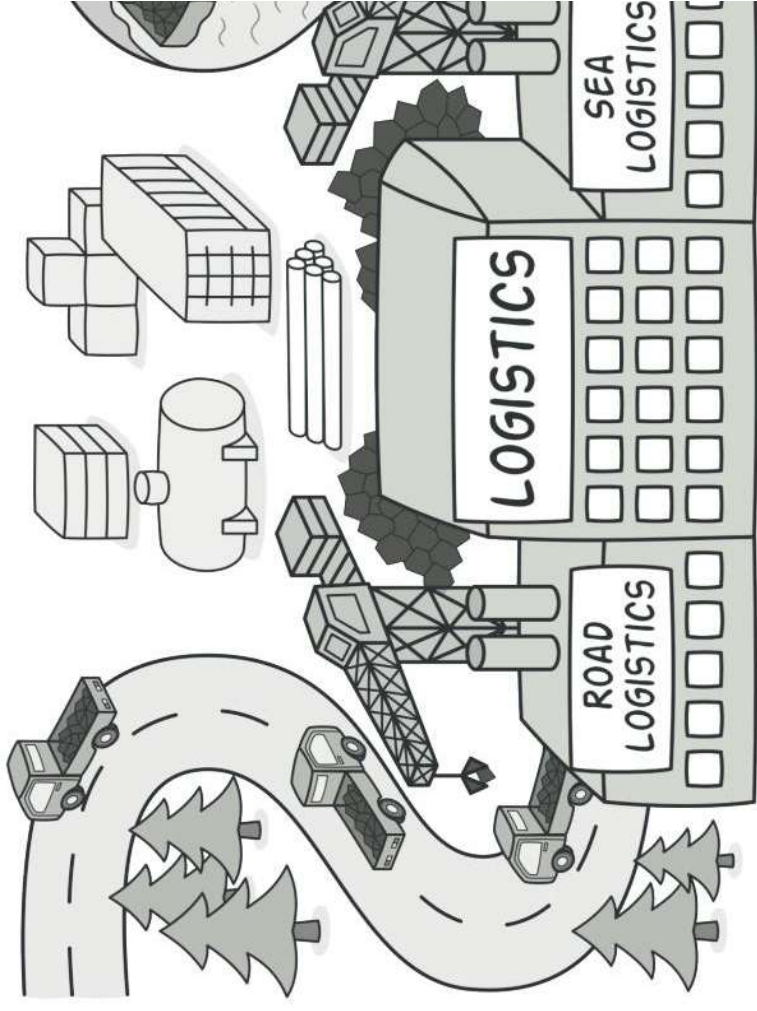


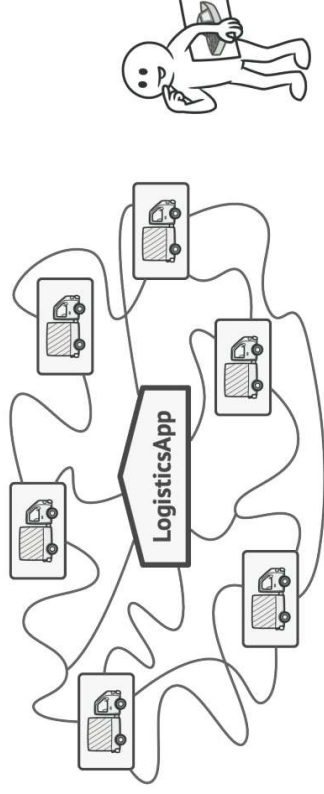
Factory Method

- O **Factory Method** é um padrão de projeto criacional que fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados.



Problema

- Imagine que você está criando um aplicativo de gerenciamento de logística. A primeira versão do seu aplicativo só pode lidar com transporte por caminhões, então a maior parte do seu código reside na classe Truck.



Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

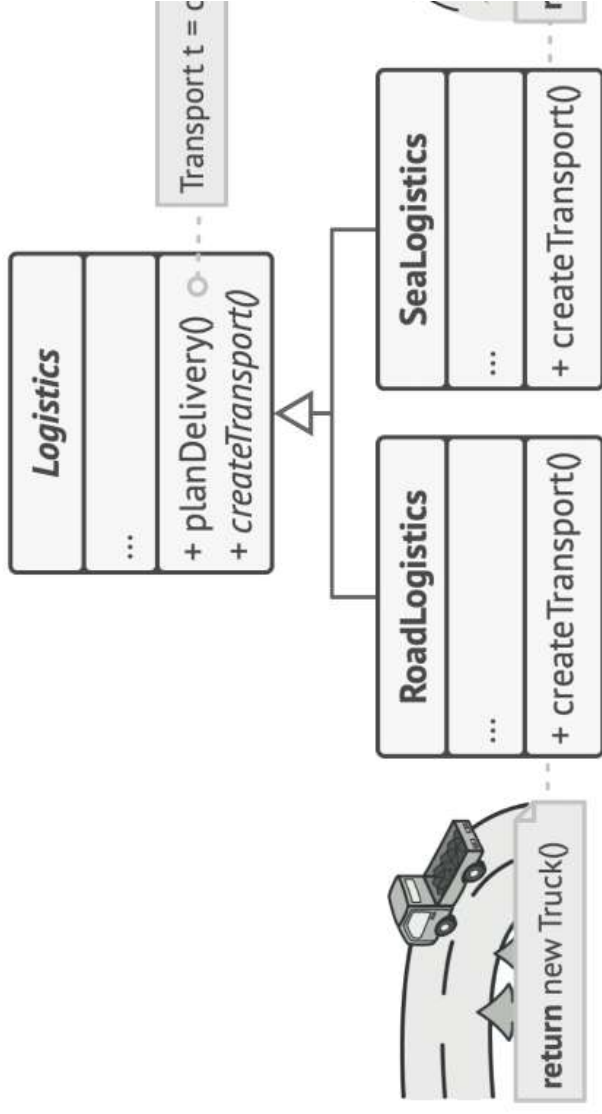
Problema

- Depois de um tempo, seu aplicativo se torna bastante popular. Todos os dias você recebe dezenas de solicitações de emprego de **transporte marítimo** para incorporar a logística marítima ao aplicativo.
- Ótima notícia, certo? Mas e o código? Atualmente, a maior parte do seu código está acoplada à classe Truck. **Adiciona Ships ao aplicativo exigiria alterações em toda a base de código.** Além disso, se mais tarde você decidir **adicionar outro tipo de transporte ao aplicativo**, provavelmente precisará refazer todas essas alterações.

- O padrão Factory Method sugere que você **substitua as chamadas diretas de construção de objetos** (usando o operador new) por chamadas a um método de fábrica especial.
- Não se preocupe: os objetos ainda são criados por meio do operador new, mas ele está sendo chamado de dentro do método de fábrica. Os objetos retornados por um método de fábrica são frequentemente chamados de produtos.

Solução

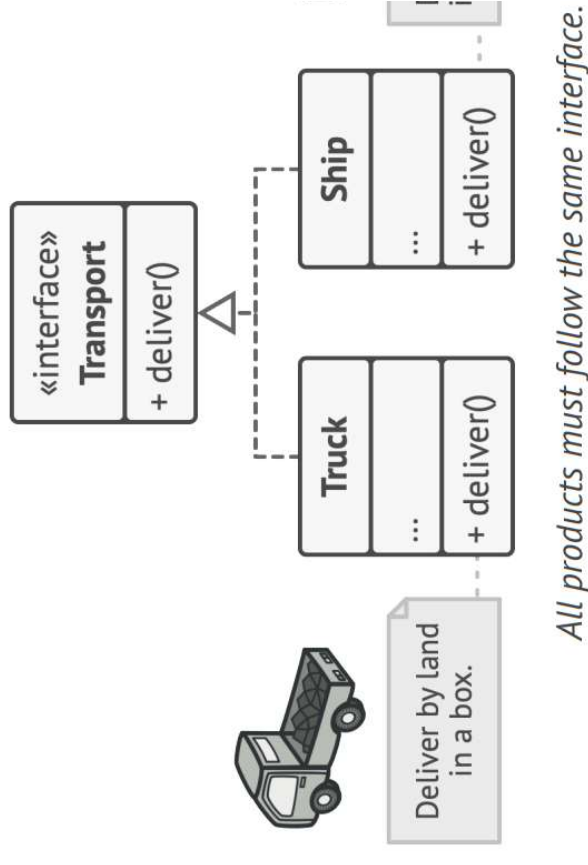
- À primeira vista, essa mudança pode parecer sem sentido: apenas movemos a chamada do construtor de uma parte do programa para outra. No entanto, considere o seguinte: **agora você pode sobrescrever o método de fábrica em uma subclasse e alterar a classe de produtos que estão sendo criados pelo método.**



Subclasses can alter the class of objects being returned by the factory method.

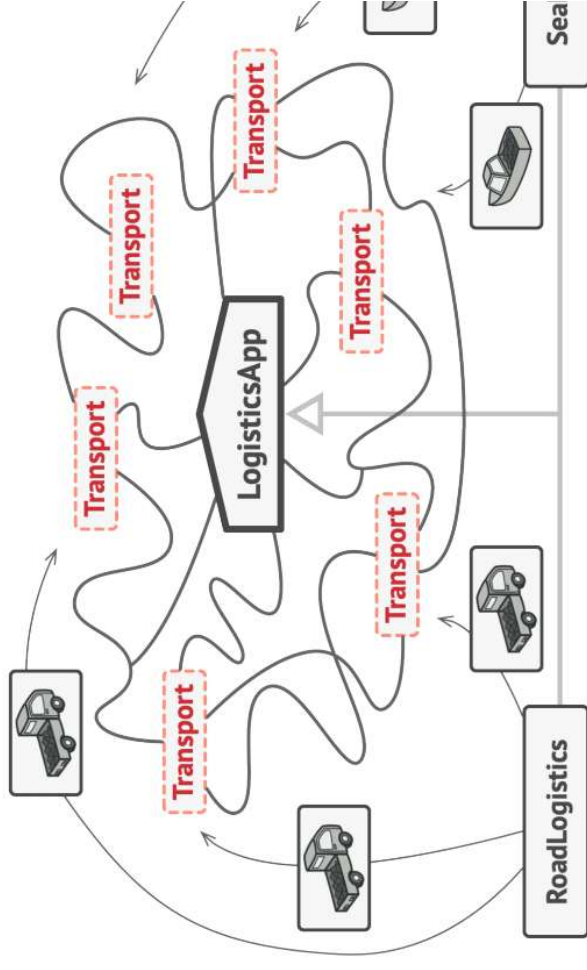
Solução

- Por exemplo, as classes Truck e Ship devem implementar a interface Transport, que declara um método chamado deliver. Cada classe implementa esse método de forma diferente: caminhões entregam carga por terra, navios entregam carga por mar. O método de fábrica na classe RoadLogistics retorna objetos de caminhão, enquanto o método de fábrica na classe SeaLogistics retorna navios.



Vantagens

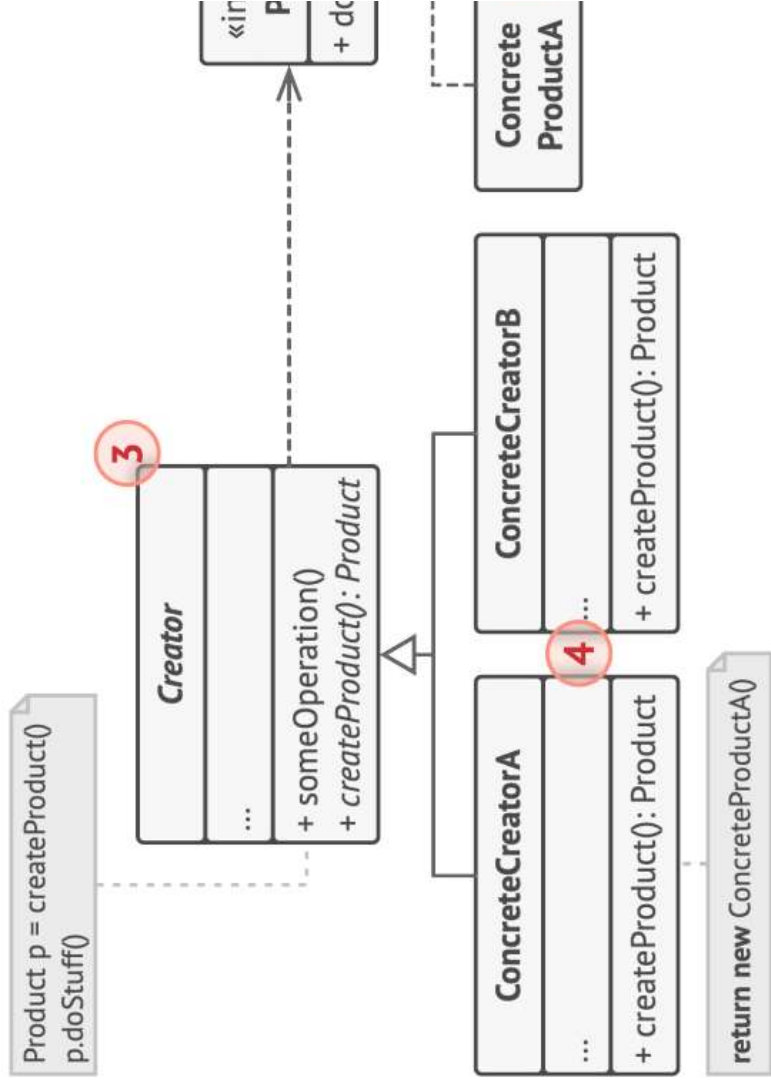
- O código que usa o método de fábrica (frequentemente chamado de código cliente) não vê diferença entre os produtos reais retornados por várias subclasses. O cliente trata todos os produtos como Transporte abstrato.



As long as all product classes implement a common interface, pass their objects to the client code without breaking it

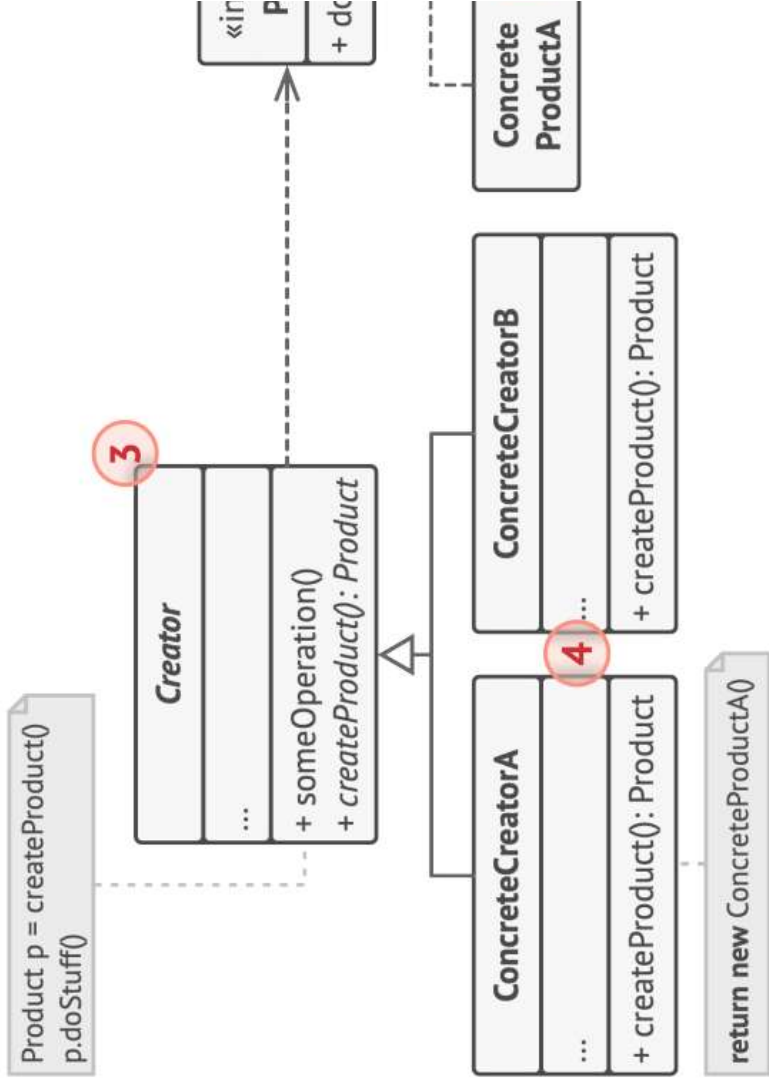
Estrutura

- O **Product** declara a interface, que é comum a todos os objetos que podem ser produzidos pelo criador e suas subclasses.



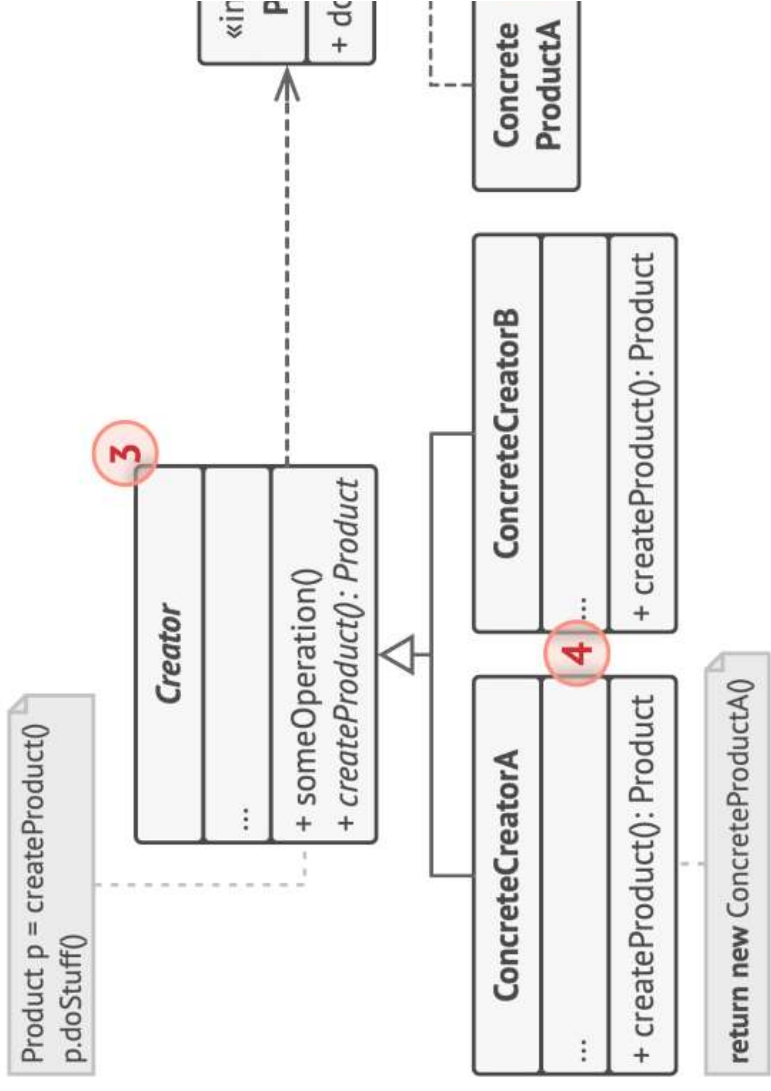
Estrutura

- Os produtos concretos são diferentes implementações da interface do produto.



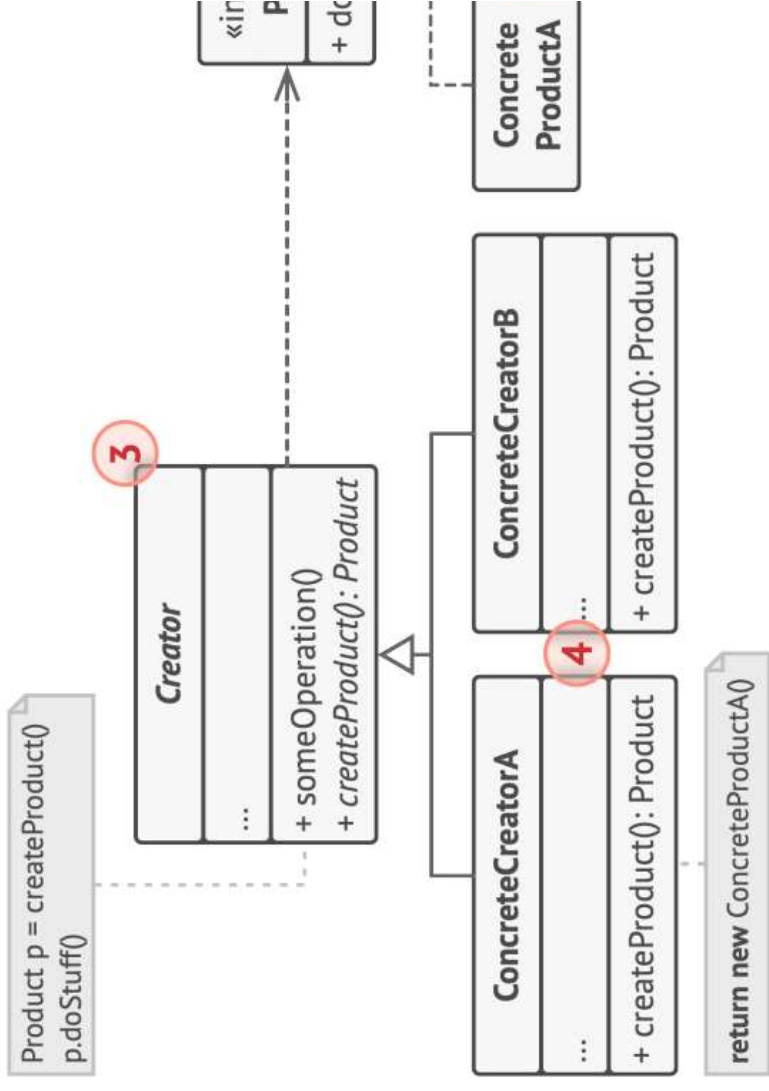
Estrutura

- A classe **Creator** declara o método de fábrica que retorna novos objetos de produto. É importante que o tipo de retorno deste método corresponda à interface do produto.



Estrutura

- Os ConcreteCreators substituem o método de fábrica base para que ele retorne um tipo de produto diferente.



Pseudocode

```
1 // The creator class declares the factory method that must
2 // return an object of a product class. The creator's subclasses
3 // usually provide the implementation of this method.
4 class Dialog is
5     // The creator may also provide some default implementation
6     // of the factory method.
7     abstract method createButton():Button
8
9     // Note that, despite its name, the creator's primary
10    // responsibility isn't creating products. It usually
11    // contains some core business logic that relies on product
12    // objects returned by the factory method. Subclasses can
13    // indirectly change that business logic by overriding the
14    // factory method and returning a different type of product
15    // from it.
16    method render() is
17        // Call the factory method to create a product object.
18        Button okButton = createButton()
19        // Now use the product.
20        okButton.onClick(closeDialog)
21        okButton.render()
```

Pseudocode

```
24 // Concrete creators override the factory method to change the
25 // resulting product's type.
26 class WindowsDialog extends Dialog is
27     method createButton():Button is
28         return new WindowsButton()
29
30 class WebDialog extends Dialog is
31     method createButton():Button is
32         return new HTMLButton()
33
34
35 // The product interface declares the operations that all
36 // concrete products must implement.
37 interface Button is
38     method render()
39     method onClick(f)
40
41 // Concrete products provide various implementations of the
42 // product interface.
```

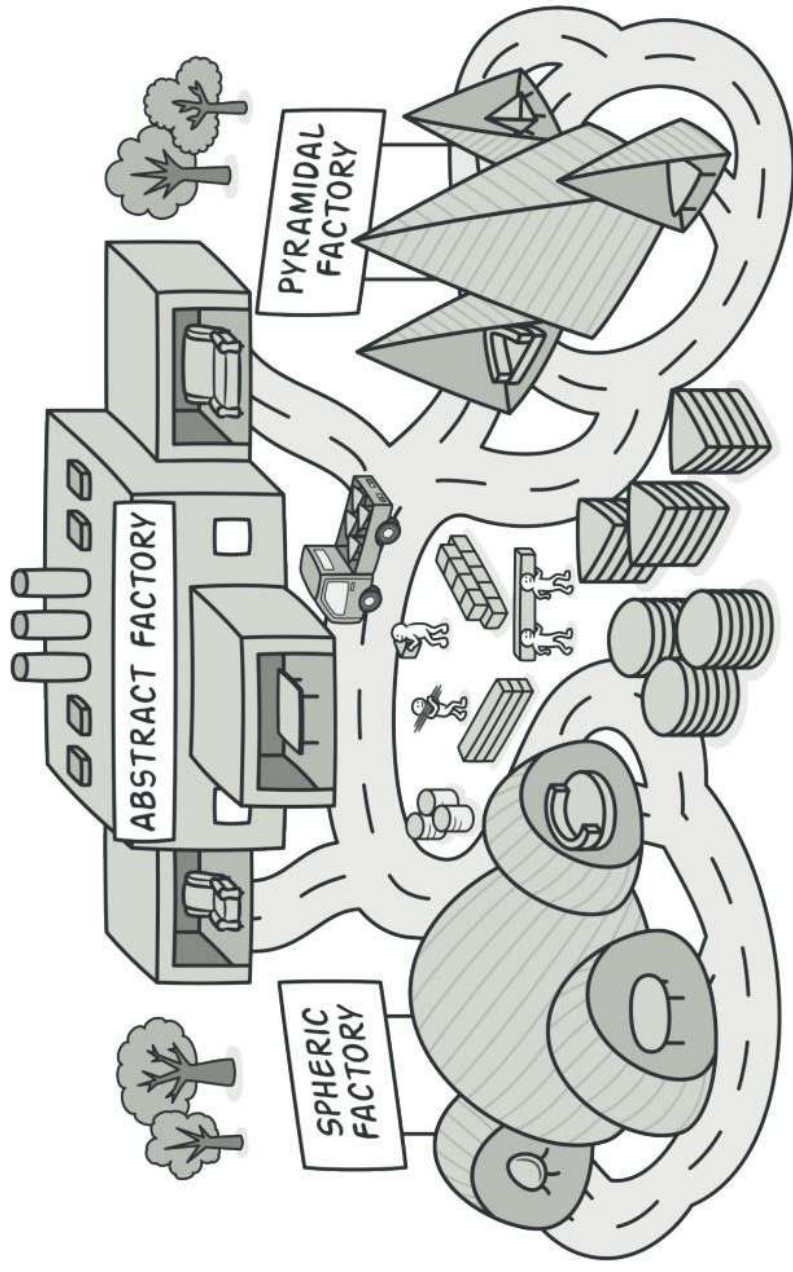
```
43 class WindowsButton implements Button
44     method render(a, b) is
45         // Render a button in Windows s
46     method onClick(f) is
47         // Bind a native OS click event
48
49 class HTMLButton implements Button
50     method render(a, b) is
51         // Return an HTML representation
52     method onClick(f) is
53         // Bind a web browser click eve
```

Pseudocode

```
56 class Application is
57   field dialog: Dialog
58
59   // The application picks a creator's type depending on the
60   // current configuration or environment settings.
61   method initialize() is
62     config = readApplicationConfigFile()
63
64     if (config.OS == "Windows") then
65       dialog = new WindowsDialog()
66     else if (config.OS == "Web") then
67       dialog = new WebDialog()
68     else
69       throw new Exception("Error! Unknown operating system.")
70
71   // The client code works with an instance of a concrete
72   // creator, albeit through its base interface. As long as
73   // the client keeps working with the creator via the base
74   // interface, you can pass it any creator's subclass.
```

```
75   method main() is
76     this.initialize()
77     dialog.render()
```

- Use quando você não souber de antemão os tipos e dependências exatos dos objetos com os quais seu código deve trabalhar.
- Use quando quiser fornecer aos usuários da sua biblioteca framework uma maneira de estender seus componentes internos.
- Use quando quiser economizar recursos do sistema reutilizando objetos existentes em vez de recriá-los a cada

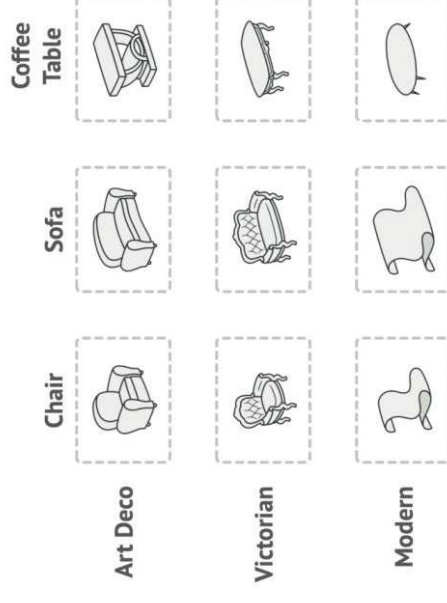


ABSTRACT FACTORY

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.

Problema

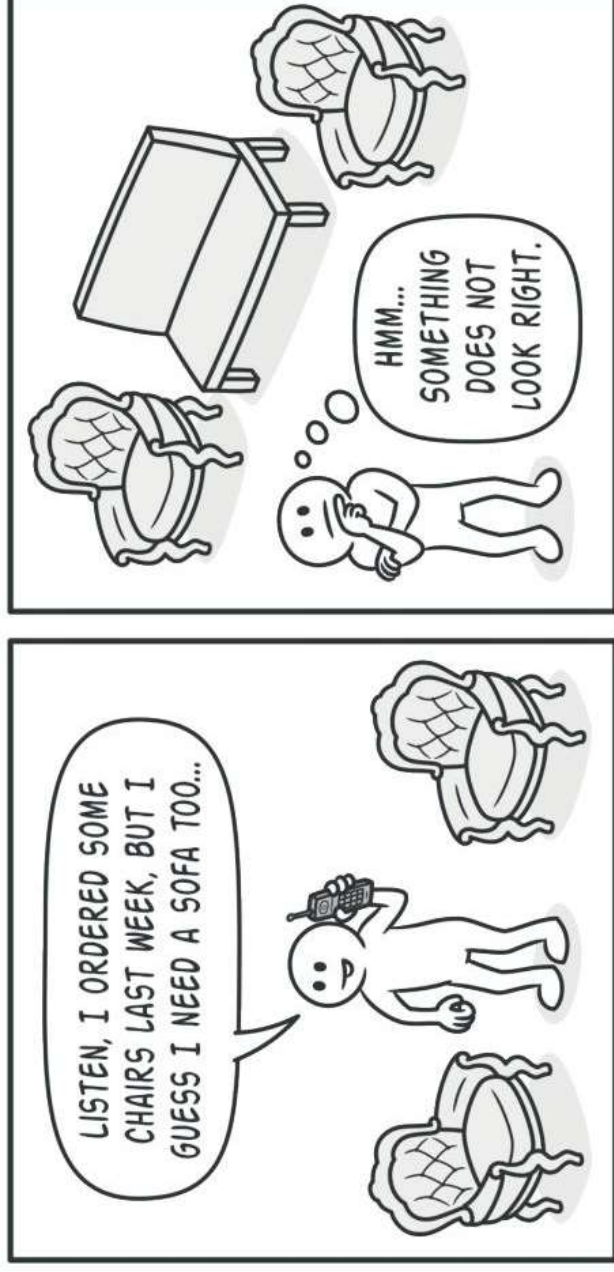
- Imagine que você está criando um simulador de loja de móveis. Seu código consiste em classes que representam:
 - Uma família de produtos relacionados, por exemplo: Cadeira + Sofa + Mesa de centro.
 - Existem diversas variantes desta família. Por exemplo, os produtos Cadeira+ Sofá + Mesa de Centro estão disponíveis nestas variantes: Moderna, Vitoriana, Art Déco.



Product families and their variants.

Problema

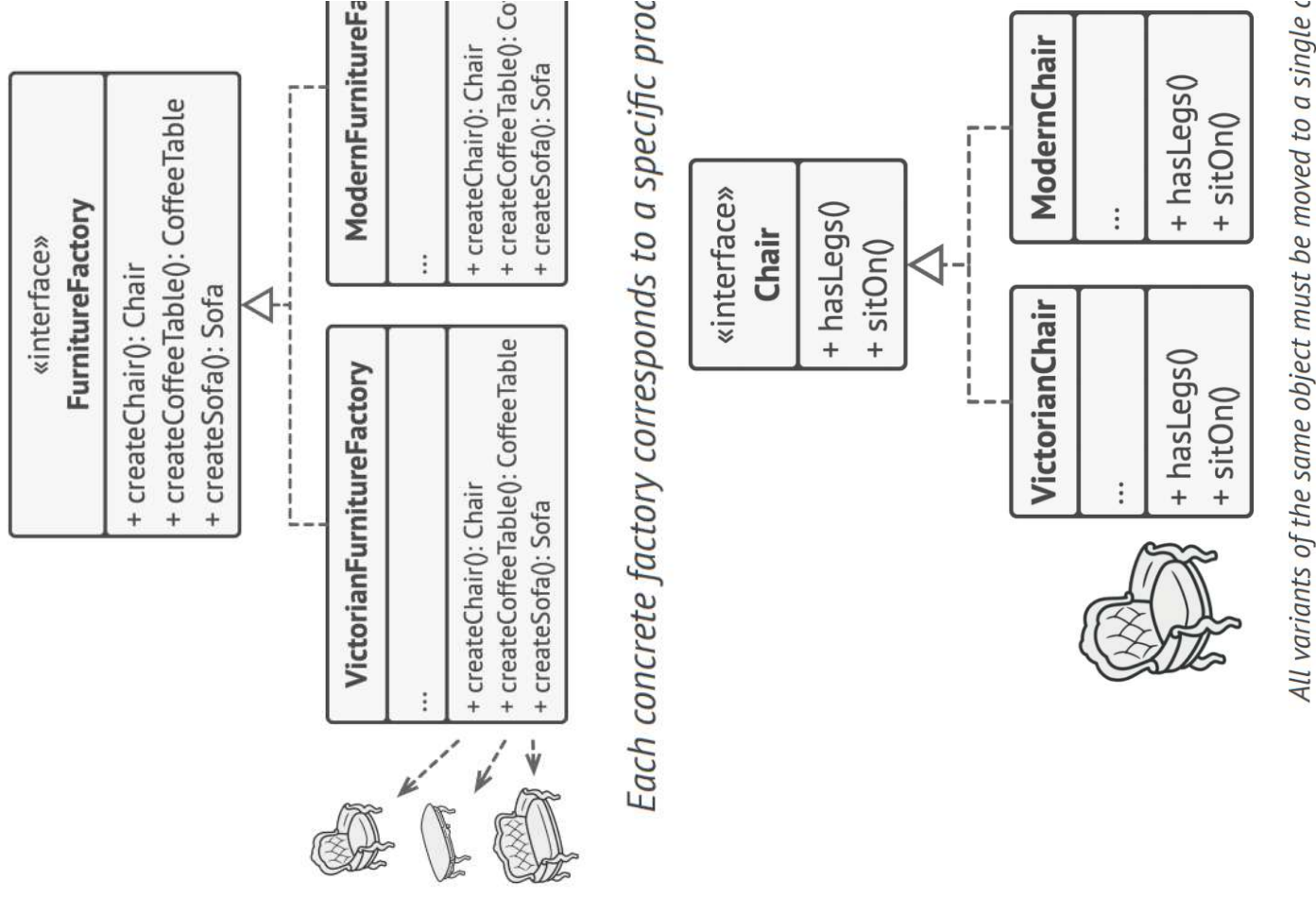
- Garantir que todos os móveis sejam do mesmo tipo após definido o tipo



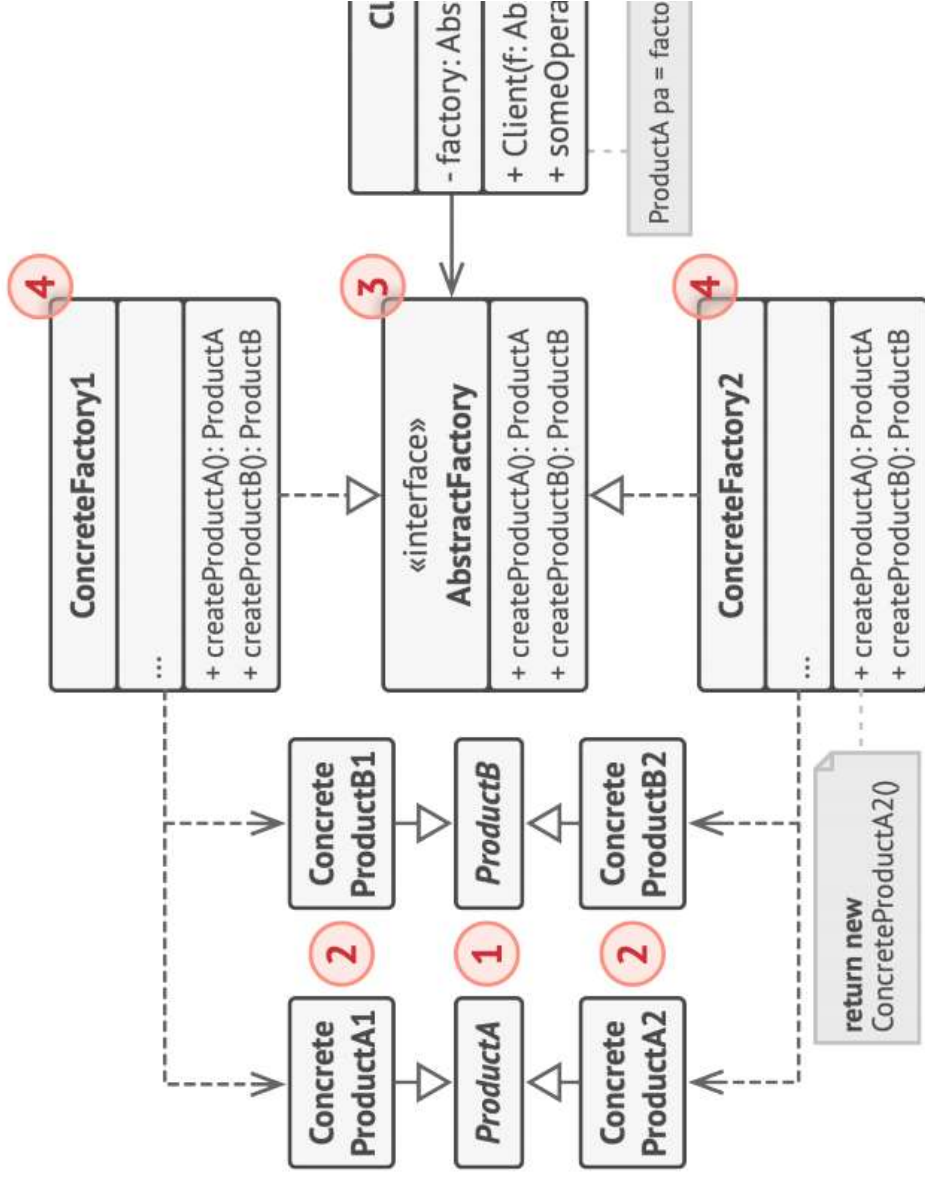
A Modern-style sofa doesn't match Victorian-style chairs.

Solução

- O próximo passo é declarar a Fábrica Abstrata — uma interface com uma lista de métodos de criação para todos os produtos que fazem parte da família de produtos.
- Esses métodos devem retornar tipos abstratos de produtos.



- A escolha da classe é feita uma vez.
- Não existe uma cadeia de IFs
- A AbstractFactory é basicamente um gerenciador de ConcreteFactory



- Use a Abstract Factory quando seu código **precisar funcionar com várias famílias de produtos relacionados**, mas você não quiser que ele dependa das classes concretas desses produtos — elas podem ser desconhecidas antecipadamente ou você simplesmente deseja permitir extensibilidade futura.
- Por exemplo, se no nosso trabalho anterior de navegação fosse usado para um jogo então cada ambiente diferente teríamos modelos compatíveis em todas as fábricas quando o nível está sendo jogado

Trabalho 6: Fábricas

- **Alterar o trabalho 5 para usar fábricas**

- Torne o projeto orientado a objetos
- Faça um tipo de fábrica para cada grupo de coisas similares, por exemplo
 - Origem e destino (podemos ter um destino dinâmico depois)
 - Caminho só horizontal e vertical e outro que permite diagonal
 - Obstáculos (quais tipos?)
 - Grid (quais tipos?)
 - Algoritmos (quais tipos?)
 - Etc

- Note que temos um conjunto de coisas que estão ligadas ao grid e temos outro conjunto de coisas que e algoritmos (etc), nisso entra a fábrica abstrata
- Queremos chegar no que está na figura, então prepare o código para isso.
- Outro nível do uso da fábrica é tornar toda a parte gráfica multiplataforma, não precisa fazer isso, mas p faria isso e me diga na próxima aula quando for na sua mesa ver o trabalho

- **GIT**

- Código

- **Prazo: 18/11 23:59**

